Recommenders, Embeddings and Matrix Factorization

Machine Learning 2020
mlvu.github.io

---

part 1:
Recommender systems
aka collaborative filtering

Matrix factorization

part 2:
PCA revisited

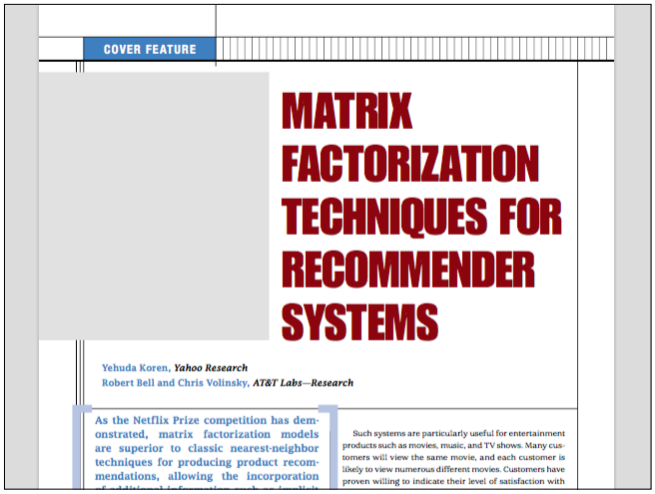
Graph models

Validating embedding models
transductive vs inductive learning

2

Today's lecture is a bit of a grab-bag of different models. The connective tissue is that these are all models for which it helps to see the data as a matrix rather than a list of vectors. We've already drawn our data as a big table or matrix, but we've mostly just operated on the individual vectors. Today, we'll see some settings where the matrix perspective actually helps us to build a good model.

We'll start with recommender systems.



COVER FEATURE

MATRIX FACTORIZATION TECHNIQUES FOR RECOMMENDER SYSTEMS

Yehuda Koren, *Yahoo Research*
Robert Bell and Chris Volinsky, *AT&T Labs—Research*

As the Netflix Prize competition has demonstrated, matrix factorization models are superior to classic nearest-neighbor techniques for producing product recommendations, allowing the incorporation

Such systems are particularly useful for entertainment products such as movies, music, and TV shows. Many customers will view the same movie, and each customer is likely to view numerous different movies. Customers have proven willing to indicate their level of satisfaction with

The exposition in the first half of the lecture will largely follow this paper.

https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf

---

NETFLIX

4

The basic, standard example of a recommender system is recommending movies to users, based on the ratings that users have already given to movies.

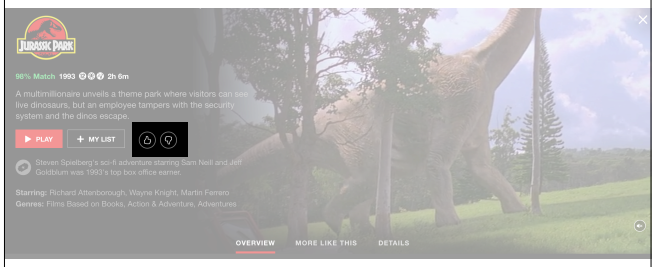
That's no accident. The modern concept of a recommender system was probably born in 2006 when Netflix, then mainly a DVD rental service, released a dataset of user/movie ratings, and offered a 1M$ prize for anybody who could improve the RMSE by 10% from Netflix's current model.

This not only sparked an interest in recommendation as a task, but also probably started the craze for machine learning competitions that later led to websites like Kaggle.

We'll use the movie task as a running example for the first half of the lecture.

explicit feedback

ask users for ratings



5

Let's look at the task, and which types of data we have available. The primary source of data is explicit user ratings: we ask users to tell us which movies they like, and hopefully, they'll oblige.

The main drawback here is that the information can be very sparse: we'll only get a few ratings per user, and some users won't give any ratings at all.

Predicting ratings based on explicit feedback is sometimes known as collaborative filtering.

---

implicit feedback

Proxies for possible ratings:

- Page views
- Wishlist
- Record cursor movement

6

To extend the ratings, we can also look for user behaviour which might be *related* to ratings. Here, we have to be a bit more careful: just because a user views a movie, doesn't mean they like it, they may just have been intrigued by the thumbnail image.

Nevertheless, if we learn from this information in the right way, it can be a treasure trove of extra signals to learn from.

---

side information (features)

about movies	about users
length	country
genre	language
actors, director	OS
synopsis	login times
awards	bio
	social media profile
	connection to other users

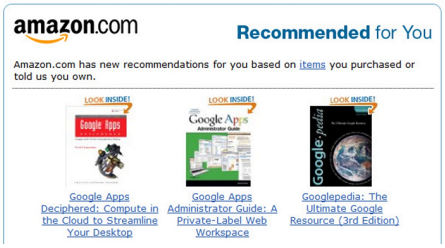
7

Finally, we have some information that is specific to just the movies and just the users. Together with the ratings, this can be a big help. Somebody who likes one Steven Spielberg adventure movie, is likely to also like another Steven Spielberg adventure movie.

This is essentially a big instance/feature matrix like we've seen already in the classic setting: one for the movies and one for the users.

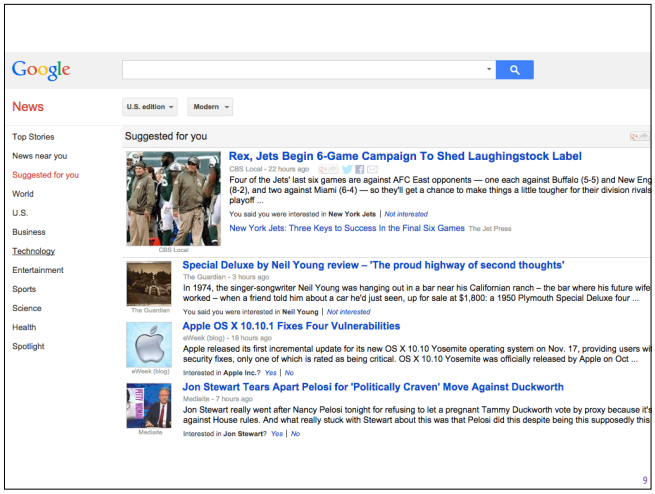
The challenge is to integrate this with the ratings, so that we can extend the relatively sparse information we get from those by generalising over the sets of users and movies.

Amazon



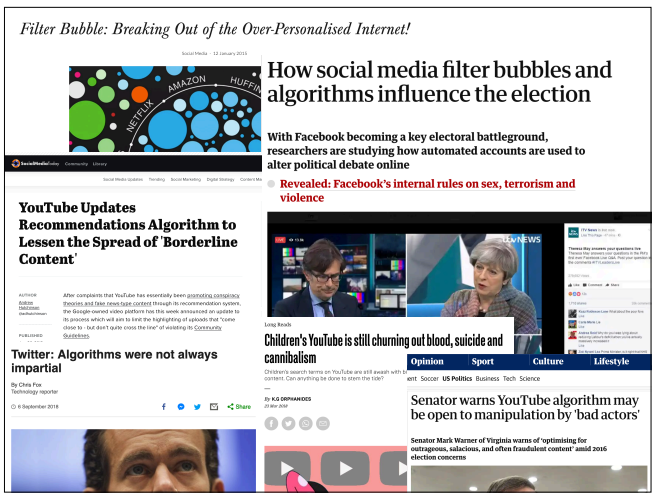
8

Movie recommendation is the canonical use case for recommender systems. Amazon was probably the first to use personalised recommendation to help users navigate their website.



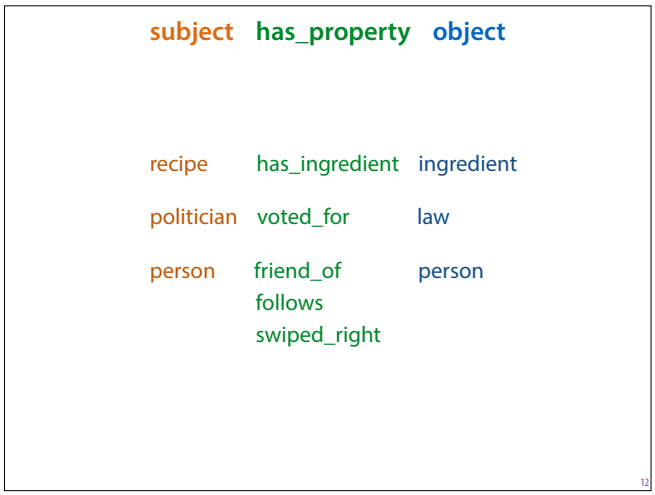
Another use case is news stories, helping people find the articles they're interested in.

Obviously, social media these days, heavily relies on recommendation. Adding recommended stories/movies/ tweets to feeds, and recommending other users to follow.



In fact recommendation algorithms are now so prevalent, that they are becoming a central component in the fabric of society: a component that is open to manipulation in the form of fake news, fake social media users a and manufactured viral content targeting young children.

In other words, it is not entirely clear at the moment whether recommendation algorithms are a force for good, or something that has grown too big for us to entirely oversee the consequences of.



In the most general sense, the **abstract task** of recommendation is applicable to any situation where you have to large sets of things (users, movies, stories, etc) and a particular relation between them. The two sets could be of the same object (for instance when predicting which two people should be friends).

the abstract task of recommendation

Given many users and many movies.

incomplete explicit ratings

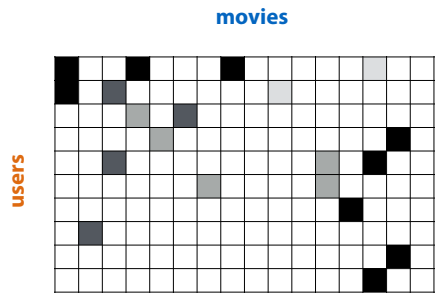
incomplete implicit ratings

(in)complete side information

predict how user u would rate item m.
Minimize the loss with the true rating

13

step 1: only explicit information



matrix: **R**

14

To keep things simple, we'll start with recommendation using only explicit information. While all the other information helps in various ways, this is the most valuable source.

For some user/movie combinations we have a rating. Most of the matrix is empty, and these values, we want to predict. For now, we will assume that the rating matrix contains real values and both positive and negative ratings are present. We will see later how to constrain these to a given interval and what to do when we have only positive ratings (like we might get from a single facebook-style "like" button).

The problem we have here is that we have no representation for the users or for the movies. The only thing we have is two big sets of "atomic" objects, when we'd like to know when two users or two movies are similar.

embedding models

Model object x by embedding vector e_x.

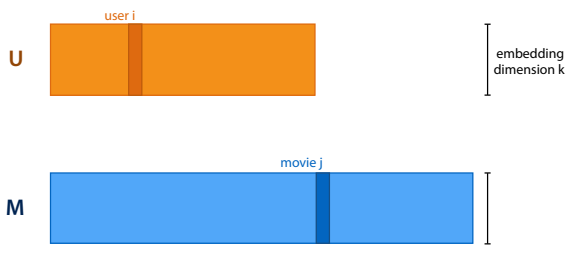
If e_x and e_y are "similar," so are x and y

Learn the parameters of $\{e_x\}$ from data.

15

We've seen this problem before, in the **word embedding** problem. There, each word was an atomic object. What we did was represent each word by its own vector, and then optimise the values of the vectors (the embedding vectors themselves were parameters of our model) to perform some downstream task (in this case predicting the context of a word).

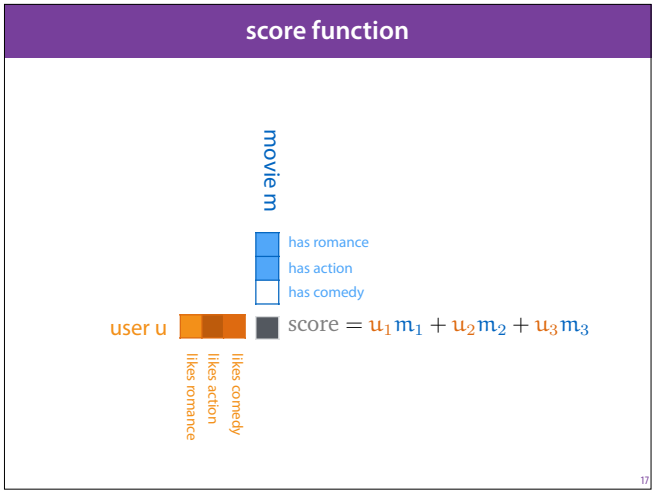
embedding models



16

We'll train a length k embedding for each user, and one for each movie, and arrange them into two matrices U and M. These are the parameters of our model.

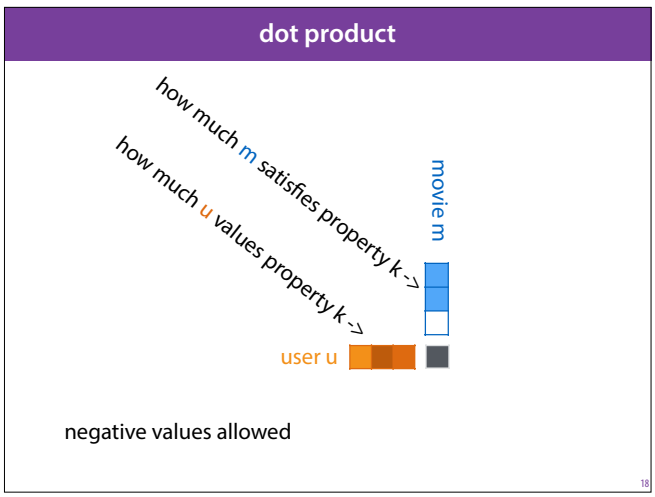
To see how to learn these values, let's imagine first what we might do if we were to set them by hand.



We can image setting the values by hand to represent various features for the users and for the movies that match each other. We might encode, for instance in one feature how much a user likes romance (negative for a strong dislike of romance and positive for a strong affinity of romance), and then encode in the corresponding movie feature, how much romance the movie contains.

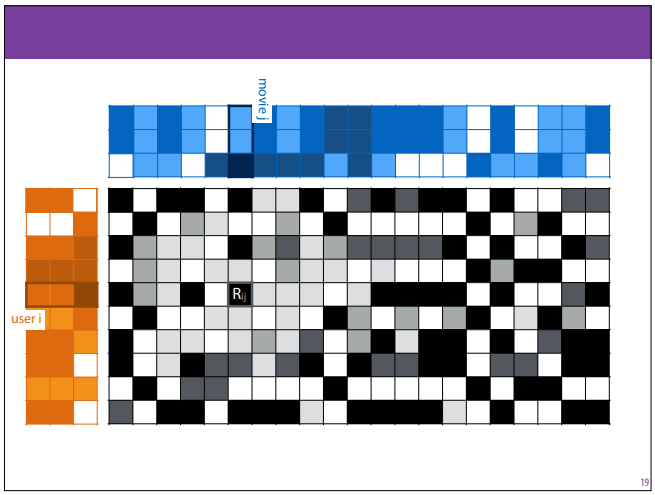
In this scenario, we can see that the dot product between the two neatly expresses how much of a match the two are: if the user loves romance and the movie contains loads of it, the romance term in the sum becomes very big. The same if both values are negative (the user hates romance and the movie is very unromantic). for mismatches, the term becomes negative and the score is brought down.

A second effect is one of *magnitude*. If the user is ambivalent to romance (i.e. the romance feature is zero), that term doesn't count towards the total (and for small values, the term contributes a little bit).



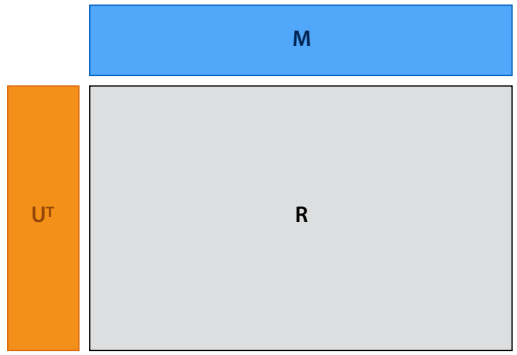
In practice, it's not feasible to enter these values for al movies and it's very difficult to gather them for users. Instead, we will learn these values, but we will use the dot product as our score function, that indicates how well a movie and a user match.

In practice, after training you *can* often recognize what particular dimensions mean, just like we saw with PCA.



Remember that in matrix multiplication A x B = C, C contains the dot products of the rows of A with the columns of B. This means that multiplying $\mathbf{U}^T$ with **M** will gives us a matrix of rating predictions for every user/movie pair.

R = $U^T M$



20

In other words, our aim is to take the rating matrix **R**, and to *decompose* it as the product as two factors: this is called **matrix factorization** (or matrix decomposition). Multiplying U and M together should produce a matrix that is as close as possible to the rain matrix we have,

---

optimisation problem

Choose **U** and **M** so that $\mathbf{U}^T\mathbf{M} \approx \mathbf{R}$

$$\arg\min_{\mathbf{U},\mathbf{M}} \|\mathbf{R} - \mathbf{U}^T\mathbf{M}\|_F^2$$
$$= \arg\min \sum_{i,j} (R_{ij} - [\mathbf{U}^T\mathbf{M}]_{ij})^2$$
$$= \arg\min \sum_{i,j} (R_{ij} - \mathbf{u}_i{}^T\mathbf{m}_j)^2$$

21

We want to decompose the given matrix **R** of ratings into the product of two smaller matrices **U** and **M** (with as little error as possible).

To qualify the error, we can use the (square of the) *Frobenius norm* of the difference between the data and the predictions. This sounds fancy, but it's the dame as flattening the matrices into vectors and computing the vector norm, or the squared error between the predictions and the data.

---

problem

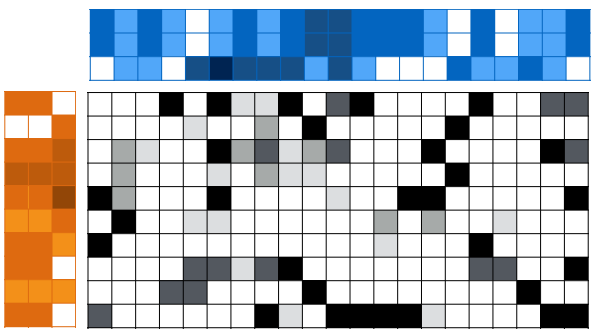
We have *a lot* of missing values in **R**

22

One problem is that **R** is not complete. For most user/movie pairs, we don't know the rating (if we did, we wouldn't need a recommender system.

---

missing values

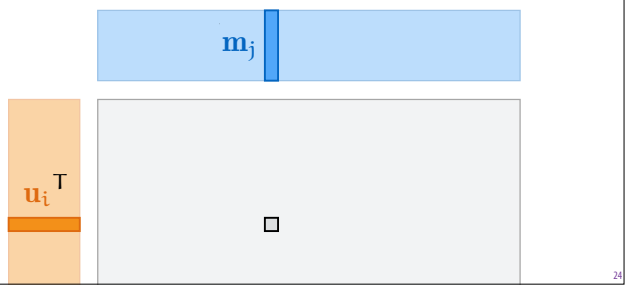


23

The matrix **R** is actually an *incomplete* matrix. We often fill in the unknown ratings with zeroes, but they are really *unknown* values.

optimise only for known ratings

$$\arg\min_{U,M} \sum_{i,j \in R_{known}} \left(R_{ij} - u_i^T m_j\right)^2$$



24

Therefore, it's sometimes better to optimise only for the *known* ratings. This is straightforward if we have both positive and negative ratings, we just compute the squared errors *only* over the known values of the matrix.

finding U and M

two options:

- alternating optimization
- gradient descent

25

alternating least squares (ALS)

Alternative to gradient descent: $\mathbf{R} = \mathbf{U}^T\mathbf{M}$.

- If we know **M**, solving for **U** is easy
- If we know **U**, solving for **M** is easy

So, starting with a random **U** and **M**.

loop:

fix **M**, compute new **U**

fix **U**, compute new **M**

26

This is a form of alternating optimization, like EM and k-means.

ALS has some computational benefits for small datasets, but in practice, gradient descent seems to be more flexible, for instance in dealing with missing values, and adding various regularizers.

stochastic gradient descent

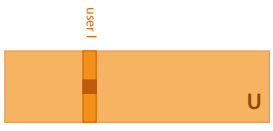
$$\frac{\partial L}{\partial u_{kl}} = \frac{1}{2}\sum_{i,j}\frac{\partial E_{ij}^2}{\partial u_{kl}} \qquad E = R - U^T M$$

$$= \frac{1}{2}\sum_{i,j} 2E_{ij}\frac{\partial E_{ij}}{\partial u_{kl}}$$

$$= \sum_{i,j} E_{ij}\frac{\partial R_{ij} - U_{\cdot i}^T M_{\cdot j}}{\partial u_{kl}}$$

$$= -\sum_{j} E_{lj}\frac{\partial U_{\cdot l}^T M_{\cdot j}}{\partial u_{kl}}$$

$$= -\sum_{j} E_{lj} M_{kj}$$



27

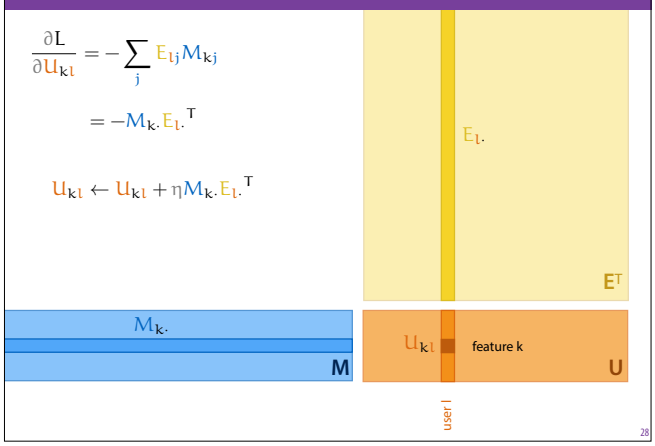
Alternating least squares is sometimes a helpful approach, but stochastic gradient descent is usually more practical.

stochastic gradient descent

$$\frac{\partial L}{\partial U_{kl}} = -\sum_j E_{lj} M_{kj}$$

$$= -M_{k\cdot} E_{l\cdot}^T$$

$$U_{kl} \leftarrow U_{kl} + \eta M_{k\cdot} E_{l\cdot}^T$$



28

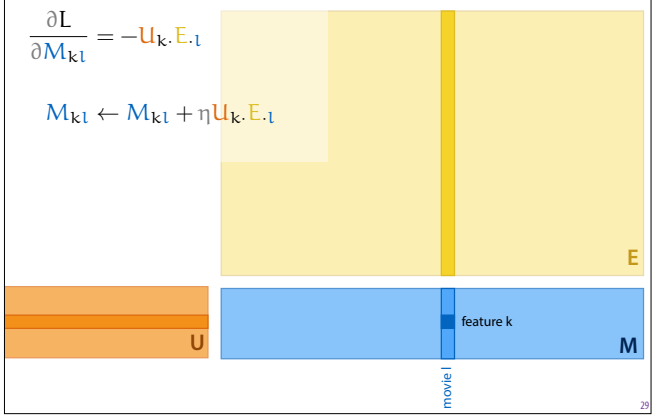
We update the parameter k for user l, by computing the error vector for user l over all movies, and taking the dot product with the k-th feature over all movies.

This means that, for instance, if a particular rating was higher than it should've been and the error for that movie is negative, and that movie has a positive value for feature k, we reduce the value U_{kl}, so that the user is matched less to that movie.

stochastic gradient descent

$$\frac{\partial L}{\partial M_{kl}} = -U_{k\cdot} E_{\cdot l}$$

$$M_{kl} \leftarrow M_{kl} + \eta U_{k\cdot} E_{\cdot l}$$

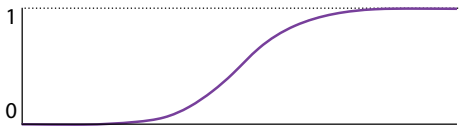


29

non-negative matrix factorization

If we have only **positive ratings** a simple approach is negative sampling:

- sample random movie user pairs as *negative* training samples (assume that users are ambivalent)
  r negative samples for each positive one, with r a hyperparameter
  apply a sigmoid to the score functions -> logistic regression



30

Instead of fitting to a particular value, we usually just want the positives to be as high as possible, and the negatives to be as low as possible.

By passing the score through a sigmoid, we can give 1 as a target value for positive examples, and 0 as a target value for negative examples. This pushes the score to positive infinity for the first and to negative infinity for the second, and it pushes harder on the examples that are closer to zero.

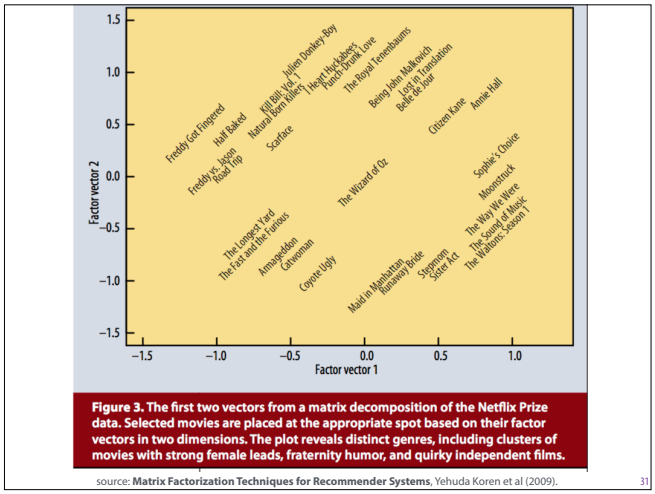


Figure 3. The first two vectors from a matrix decomposition of the Netflix Prize data. Selected movies are placed at the appropriate spot based on their factor vectors in two dimensions. The plot reveals distinct genres, including clusters of movies with strong female leads, fraternity humor, and quirky independent films.

source: **Matrix Factorization Techniques for Recommender Systems**, Yehuda Koren et al (2009).

31

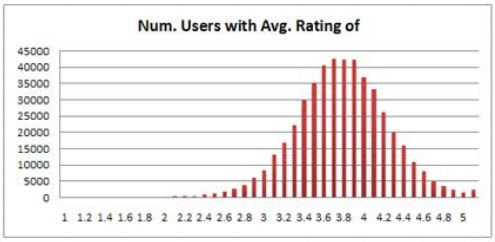
source: **Matrix Factorization Techniques for Recommender Systems**, Yehuda Koren et al (2009).

improvements

- control for user bias
- control for movie bias
- use implicit feedback
- use side information
- control for temporal bias

32

user bias



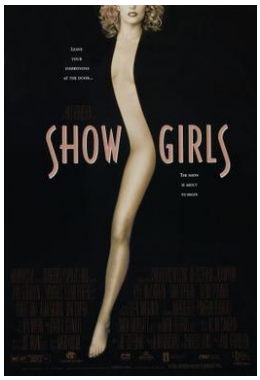
source: http://www.hackingnetflix.com/2006/10/netflix_prize_d.html

33

The average rating for each user is different. Some users are very positive, giving almost every movie 5 stars, and some give almost every movie less than 3 stars.

If we can explicitly model the bias of a user, it takes some of the pressure off the matrix factorisation, which then only needs to predict how much a user will deviate from their average rating for a particular movie.

movie bias



34

The same is true for movies. Some movies are universally liked, and some are universally loathed.

biases

b: generic bias

b_i: user bias

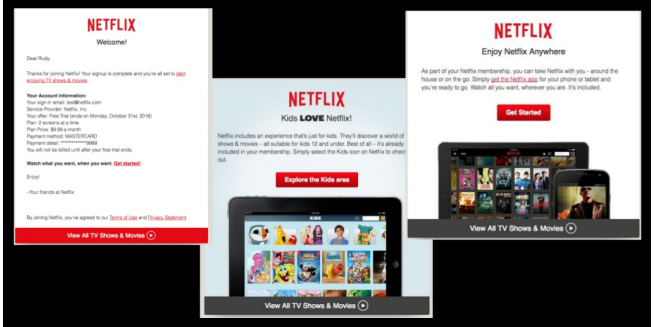
b_j: movie bias

$$P_{ij} = \mathbf{u}_i^T \mathbf{m}_j + b_i + b_j + b$$

35

We model biases by a simple additive scalar (which is learned along with the embeddings): one for each user, one for each movie, and one general bias over all ratings.

cold start problem



source: https://www.slideshare.net/AmazonWebServices/aws-reinvent-2016-how-netflix-achieves-email-delivery-at-global-scale-with-amazon-ses-mbl204

36

One big problem in recommender systems is the **cold start problem**. When a new user joins Netflix, or a new movie is added to the database, we have no ratings for them, so the matrix factorization has nothing to build an embedding on.

I this case we have to rely on implicit feedback and side information.

---

using implicit "likes"

Movies watched by the user, but not rated

Movies browsed by the user

Movies liked by similar users

Movies hovered over with the mouse

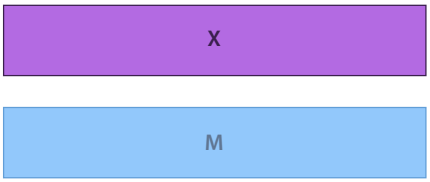
N(i): all movies implicitly liked by user i

37

---

new embedding X

$$\mathbf{u}_i^{\text{imp}} = \sum_{j \in N(i)} \mathbf{x}_j$$



38

We add a separate movie embedding x, and then compute a second user embedding which is the sum of the x-embeddings of all the movies user x has implicitly "liked" (i.e. in some way been associated with).

It can also help to normalise this (see the paper).

$$P_{ij} = (\mathbf{u}_i + \mathbf{u}_i^{\text{imp}})^T \mathbf{m}_j + b_i + b_j + b$$

39

We then add the implicit-feedback embedding to the existing one before computing the dot product.

using side information

User features: age, login, browser resolution, social media

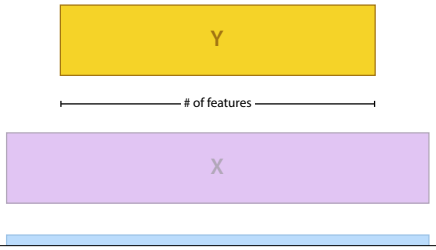
A(u): all features of user u

40

To simplify things, we'll assume all features are binary categories: the feature applies or it doesn't.

new embedding Y

$$\mathbf{u}_i^{\text{side}} = \sum_{f \in A(i)} \mathbf{y}_f$$



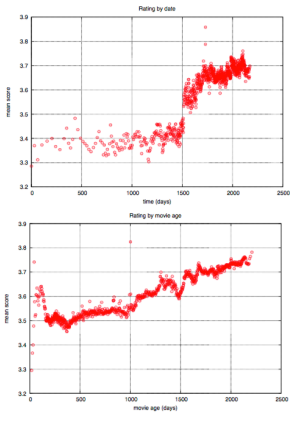
41

We then assign each feature an embedding, and sum over all features that apply to the user, creating a third user embedding.

$$P_{ij} = (\mathbf{u}_i + \mathbf{u}_i^{\text{imp}} + \mathbf{u}_i^{\text{side}})^T \mathbf{m}_j + b_i + b_j + b$$

42

time



source: Collaborative Filtering with Temporal Dynamics, Yehuda Koren

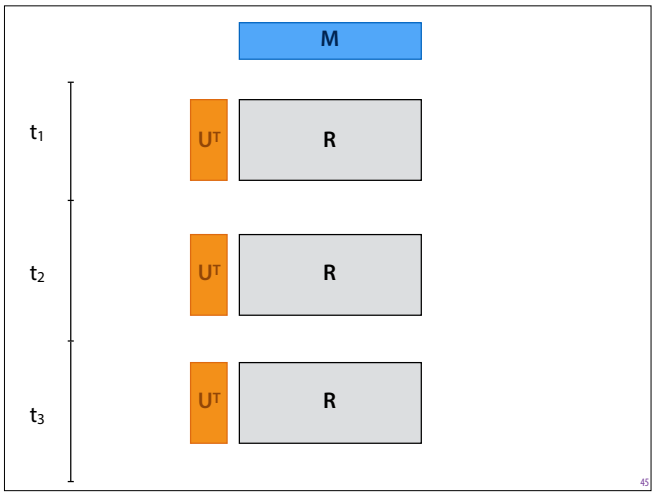
43

The Netflix data is not stable over time. It covers about 7 years, and in that time many things have changes. The most radical change comes about four years in, when Netflix changed the meaning of the ratings in words (these appeared in mouseover when you hovered over the ratings). Specifically, they changed the one-star rating from "I didn't like it" to "I hated it". Since people are less likely to say that they hate things, the average ratings increased.

Similarly, if you look at how old a movie is, you see a positive relation to the average rating. Generally, people who watch a really old movie will likely do so because they know it, and want to watch. Whereas for new movies, more people are likely to be swayed by novelty and advertising. This means that new movies have a temporal bias for lower ratings.

$$P_{ij}(t) = u_i(t)^T m_j + b_i(t) + b_j(t) + b$$

The solution is to make the biases, and the user embeddings time dependent. For the movies we make only the bias time dependent, since the properties of the movie itself stay the same. For user embeddings, we can actually make the embeddings time dependent, since user tastes may change over time.



A very practical way to do this is just to cut time into a small number of chunks and learn a separate embedding for each chunk.

Note that all the matrices stay the same size. There are just fewer ratings in **R**.

summary

When your task consists of linking one large set of things to another large set of things, based on sparse examples, and little intrinsic information, **matrix factorization** may be appropriate.

Extend your models with biases, regularizers, implicit likes, side information and temporal dynamics.

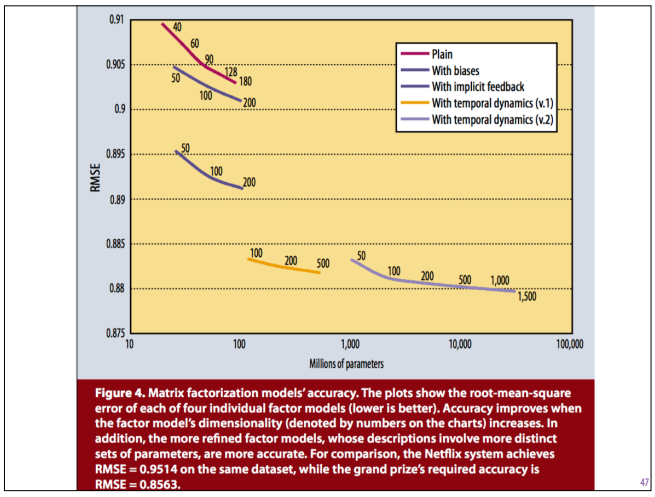
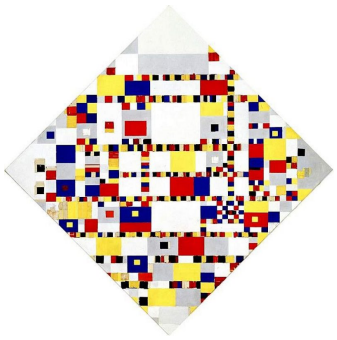


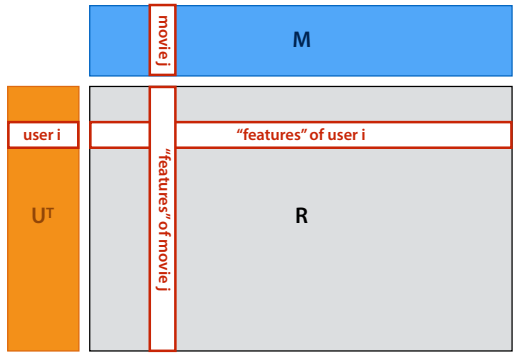
Figure 4. Matrix factorization models' accuracy. The plots show the root-mean-square error of each of four individual factor models (lower is better). Accuracy improves when the factor model's dimensionality (denoted by numbers on the charts) increases. In addition, the more refined factor models, whose descriptions involve more distinct sets of parameters, are more accurate. For comparison, the Netflix system achieves RMSE = 0.9514 on the same dataset, while the grand prize's required accuracy is RMSE = 0.8563.

Here is how the different additions to the basic matrix factorization ultimately served to reduce the RMSE to the point that won the authors the netflix prize.

break



48



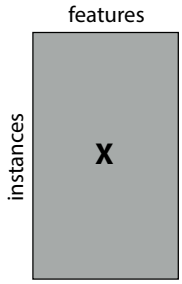
49

Matrix factorization is a bit of a strange setting, where instead of instances with features, we have two types of instances, with the links between them determining what we know about each item and each user.

We can think of this as two traditional data matrices in one: if we consider the users as features, then the movies are a big set of binary features. If we consider the movies as instances, then which users they are liked are their features.

---

classic ML: factoring the data matrix



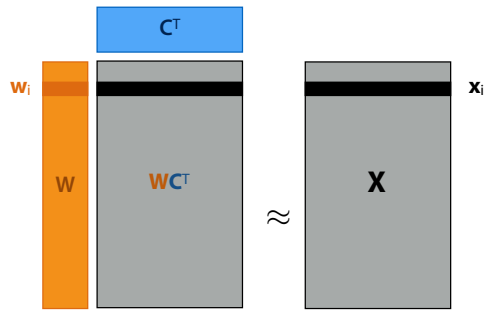
we'll assume X is mean-subtracted

see also: http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/

50

In the classical machine learning setting, our data can also be seen as a matrix (usually with an instance per row, and a feature per column. What would happen if we apply matrix factorization to this matrix?

NB: In the following we'll assume that the data have been mean-subtracted (the mean over all rows has been subtracted from each row).

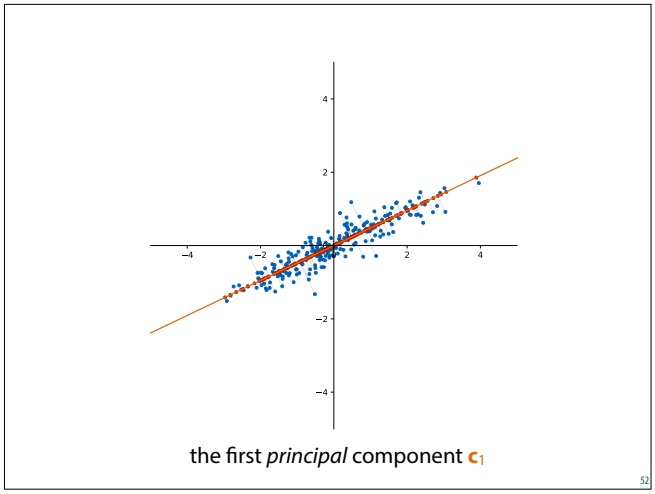


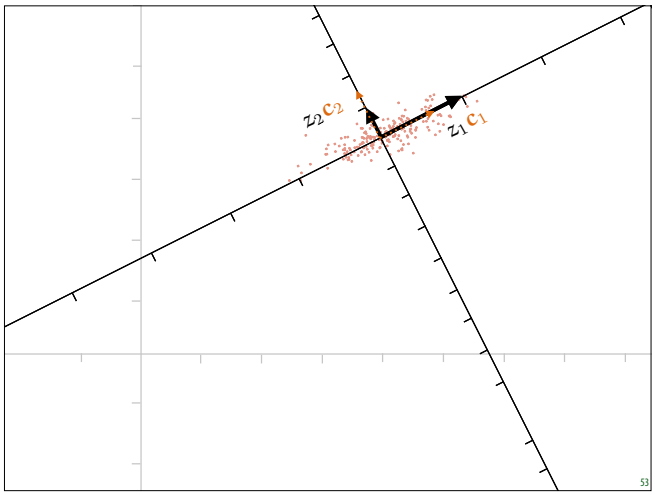
w_i: a low-dimensional embedding, from which x_i can be reconstructed with low MSE.

51

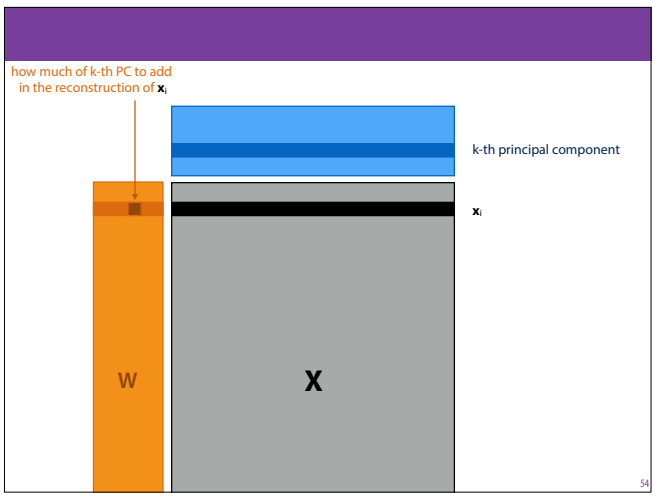
If we apply the same principle as we did with the recommender system, we are looking for two matrices, W and C: the first containing "embeddings" for our instances and the second "embeddings" for our features, such that their dot product reconstructs, as much as possible, the value of a particular feature for a particular matrix.

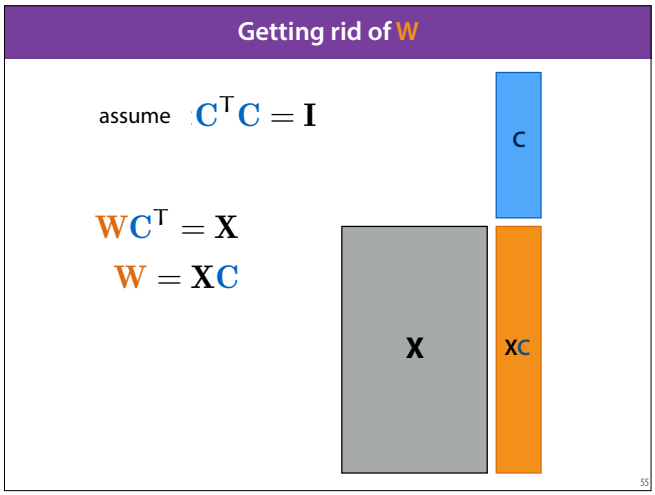
This idea, dimensionality reduction by minimizing the squared error loss should remind us of PCA.





If we do that, we still get a rebasing of the data. It turns out, that this gives us a whitening of the data. If we take the **c** vectors as an orthonormal basis, and then stretch by the corresponding z values, we get a coordinate system in which the new data looks like a standard normal distribution (or as much like a standard normal distribution as we can achieve with a linear transformation).





We can make it equivalent to PCA, by assuming that the columns of C are linearly independent. In this case, we can rewrite W in terms of C,and reduce the parameters of the model to just the “feature embeddings”.

from matrix factorisation to PCA

$$\arg\min_{\mathbf{W},\mathbf{C}} ||\mathbf{X} - \mathbf{W}\mathbf{C}^\mathsf{T}||^2$$

$$\arg\min_{\mathbf{C}} ||\mathbf{X} - \mathbf{X}\mathbf{C}\mathbf{C}^\mathsf{T}||^2$$

such that: $\mathbf{C}^\mathsf{T}\mathbf{C} = \mathbf{I}$

solution:
- eigendecomposition
- singular value decomposition
- gradient descent / alternating least squares

This gives us a constrained optimisation problem that is equivalent to PCA. The columns of $\mathbf{C}$ (describing a mixture over the features) are the principal components.

incomplete PCA

$$\arg\min_{\mathbf{C}} \sum_{i,j \in \text{known}} (X_{ij} - [\mathbf{W}^\mathsf{T}\mathbf{C}]_{ij})^2$$

Dimensionality reduction/data completion

Apparently a much trickier problem than complete PCA

This perspective e allows us to modify the PCA objective with tricks we've seen in other settings. For instance, if our data has missing values, we can focus the optimization only on the known values, giving us a mixture of dimensionality reduction and data completion.

L2 regulariser

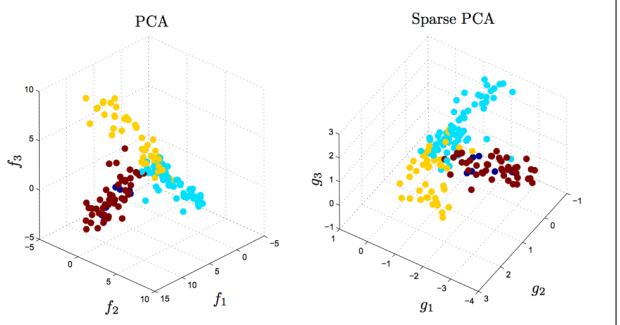
$$\arg\min_{\mathbf{W},\mathbf{C}} ||\mathbf{X} - \mathbf{W}\mathbf{C}^\mathsf{T}||_\mathsf{F}^{\ 2} + \gamma_1 \sum_i ||\mathbf{w}_i||_2^{\ 2} + \gamma_2 \sum_j ||\mathbf{c}_j||_2^{\ 2}$$

We can also add a regularizer to constrain the complexity of our embeddings.

L1 regulariser

$$\arg\min_{\mathbf{W},\mathbf{C}} ||\mathbf{X} - \mathbf{W}\mathbf{C}^\mathsf{T}||_\mathsf{F}^{\ 2} + \gamma_1 \sum_i ||\mathbf{w}_i||_1^{\ 2} + \gamma_2 \sum_j ||\mathbf{c}_j||_1^{\ 2}$$

An L1 regularizer, as we know (see Lecture 41 Deep Learning 1), promotes sparse models: models for which parameters are exactly zero. In this case that means that our embeddings are more likely to contain zeroes, which can make it easier to interpret the results (a plot like that in slide 30 would be more axis aligned).



source: **http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/**

60

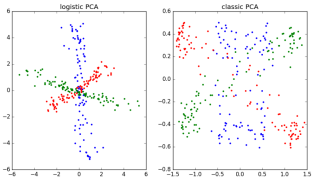
other variants

positive data: non-negative PCA

use non-negative matrix factorization

binary data: logistic PCA

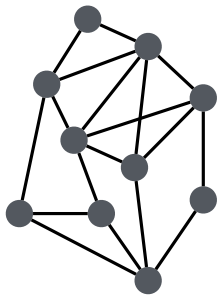
apply logistic function, use cross-entropy loss



source: **http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/**

61

graphs



social graphs

protein interaction

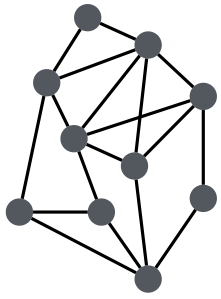
traffic networks

knowledge graphs

62

Graphs are an even more versatile format for capturing knowledge than matrices and tensors. Many of the most interesting datasets come in the form of graphs.

link prediction

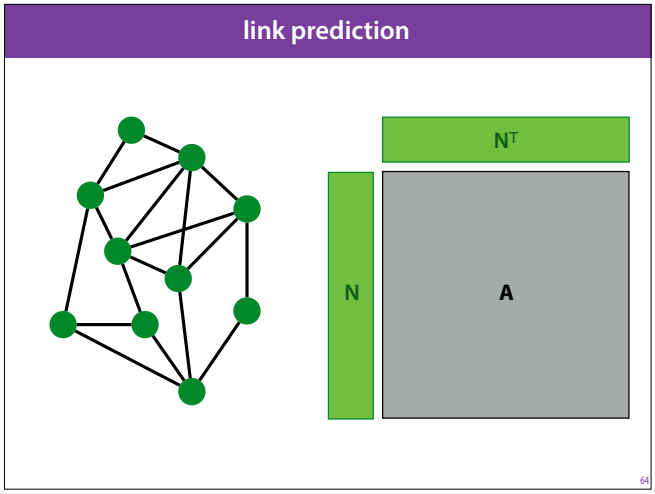


Very similar to recommendation

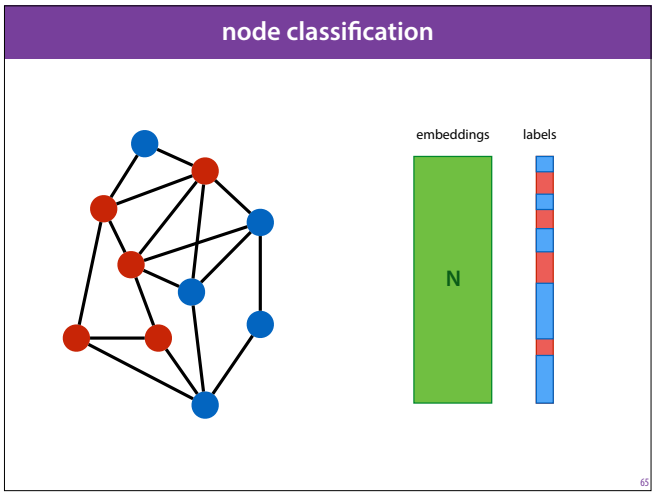
Assume the graph is incomplete, predict links.

63

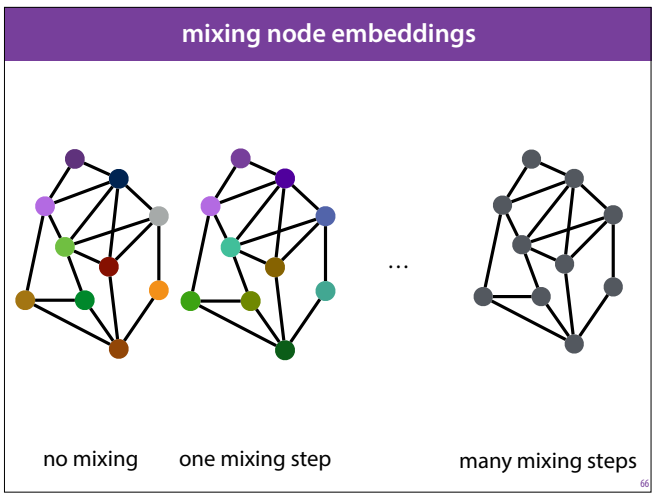
In link prediction, we assume the graph we see is incomplete (which is usually the case) and we try to predict which nodes should be linked .



We can easily treat this just like a matrix factorization problem. But it would be nice to look a little deeper into the graph.



Node classification is another task: for each node, we are given a label, which we should try to predict. If we have node embeddings, we can use those in a regular classifier, but the question is, how to we get those embeddings (and how do we make sure that they capture the required properties of the graph structure).

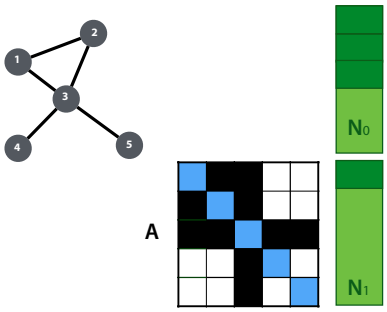


The principle we will be using to learn/refine our embedding is that of mixing embeddings. To develop out intuition, imaging that we assign a nodes random 3D embeddings (with values between 0 and 1). For the purposes of visualization, we can interpret these as RGB colors.

We then apply a mixing step: we replace each node color by the mixture (the average) of itself and its direct neighbors. At first, the embeddings express nothing but identity, each node has its own color. After one mixing step, the node embeddings express something about the local graph neighbourhood: a node that is close to many purple nodes will come slightly more purple itself.

After many mixing steps, all nodes have the same embedding, expressing only information about the entire graph. Somewhere in between we find a sweet spot where the embeddings express the node identity, but also the structure of the local graph neighborhood.

mixing node embeddings



options for **A**:

- **A** + **I**, row-normalised (works on most graphs)
- **A** + **I**, symmetric normalisation (only on undirected graphs)

67

We can achieve this mixing by post-multiplying the embedding vector by the adjacency matrix: this results in the sum of the embeddings of the neighbouring nodes. We also add self-loops for every node so that the current embedding stays part of the sum.

If we sum, the embedding will blow up. with every mixing step. In order to control for this we need to normalize the adjacency matrix. If we row-normalize, we get the average over all neighbours. We can also use **symmetric normalization**, which leads to a slightly different type of mixing but only works on undirected graphs.

See this article for more details: **https://tkipf.github.io/graph-convolutional-networks/**

---

graph convolutional network (GCN)

Start with some node embeddings $\mathbf{N}_0$.

Compute a new embedding for each node: the average of its neighbor's embeddings and its own:
$\mathbf{A}\mathbf{N}_0$

Multiply by a weight matrix $\mathbf{W}$, add activation:

$$\mathbf{N}_1 = \sigma(\mathbf{A}\mathbf{N}_0\mathbf{W})$$

68

In order to make the mixing trainable, we add a a multiplication by a weight matrix. This matrix applies a linear projection to the mixed embeddings.

The sigmoid activation can also be ReLU or linear. What works best depends on the data.

---

multiple "layers"

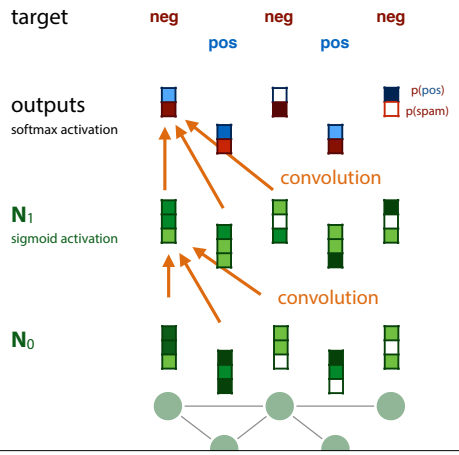
$$\mathbf{N}_1 = \sigma(\mathbf{A}\mathbf{N}_0\mathbf{W})$$
$$\mathbf{O} = \sigma(\mathbf{A}\sigma(\mathbf{A}\mathbf{N}_0\mathbf{W})\mathbf{V})$$

after k layers, the embedding mixes in information from k hops away

69

Applying this principle multiple times leads to a multi-layered structure

---

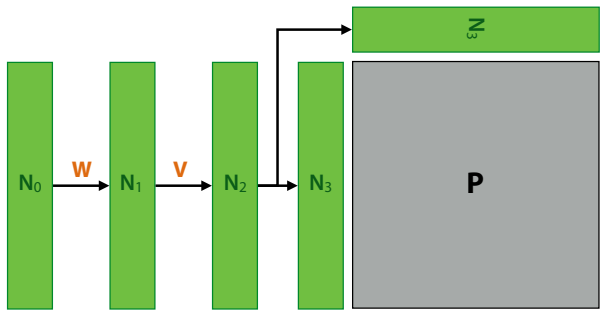


70

Here is how we do node classification with graph convolutions. We ensure that the embedding size of the last layer is equal to the number of classes (2 in this case). We then apply a softmax activation to these embeddings and interpret them as probabilities over the classes.

This gives us a full batch of predictions for the whole data, for which we can compute the loss, which we then backpropagate.

link prediction



71

When we do link prediction, we can perform some graph convolutions on our embeddings and then multiply them out to generate our predictions. We compare these to our training data, and back propagate the loss.

GCNs

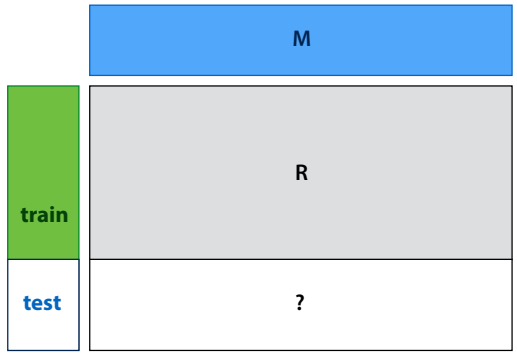
Depth is a problem: high connectivity diffuses information

Usually full-batch: no straightforward way to break up the graph into minibatches.

Pooling is not *selective*: all neighbours are mixed equally before weights are applied.
The weights do not affect which neighbours receive attention

72

validation



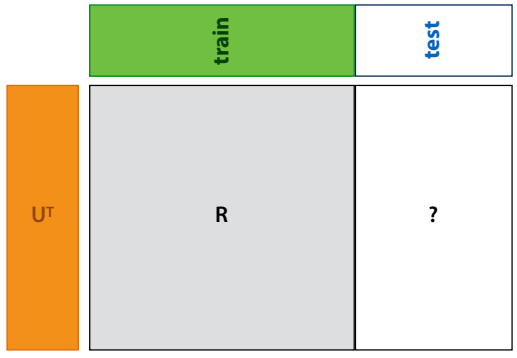
source: http://alexhwilliams.info/itsneuronalblog/2018/02/26/crossval/?mlreview

73

A final point: it is important to carefully consider our validation protocol. In other words: how do we withhold test data to train on.Let's start with recommender systems. At first, you might think that it's a good idea to just withhold some users.

However, this doesn't work: if we don't see the users during training, we won't learn embeddings for them, which means we can't generate predictions.

validation



74

The same thing happens if we randomly withhold some movies.

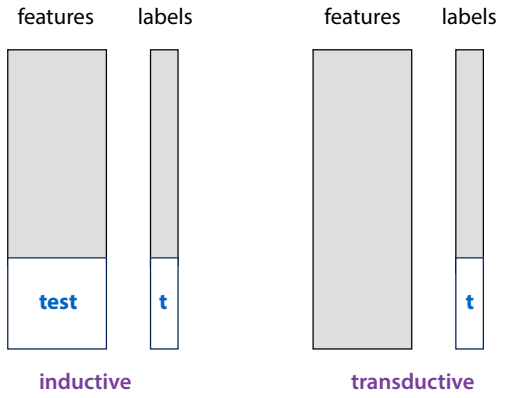
mixed features

the features of our users are their ratings over all movies

the features of our movies are their ratings from all users.

75

inductive vs transductive learning



76

This is related to the difference between **inductive** and **transductive** learning. I the transduction setting, the learning is allowed to see the features of all data, but the labels of only the training data.

inductive vs transductive learning

Embedding models only support transductive learning.
If we don't know the objects until after training, we won't have embedding vectors.

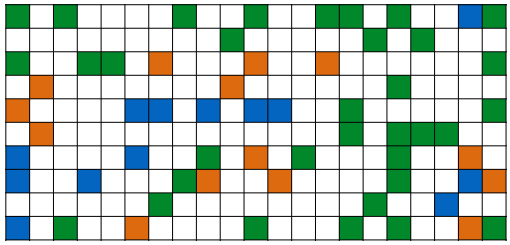
Word2Vec: we need to know the *whole* vocabulary at training time.

Recommendation: we need to know *all* users and movies.

Graph models: we need to know the *whole* graph.

77

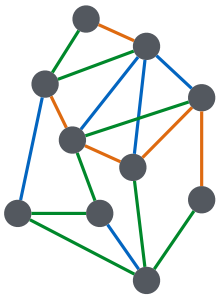
recommendation: train, validation, test



78

To evaluate our matrix factorization, we give the training algorithm all users, and all movies, **but withhold some of the ratings**.

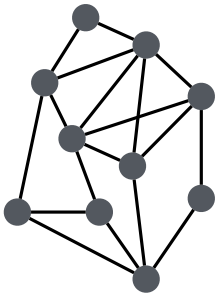
graphs: link prediction

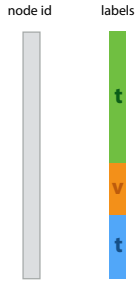


79

The same goes for the links.

graphs: node classification





80

In the case of node classification, we provide the algorithm with the whole graph, and a table linking the node ids to the labels. In this table, we withhold some of the labels.

validation: timestamps

No training on data from the future
ratings, nodes can have timestamps

All training data should come before all validation data, which should come before all test data.

81

If our data has timestamps, we should follow the advice from last lecture as well.

summary

Abstract task: recommendation
Good for any situation where you have two **large** sets of objects, with relations between them

Matrix factorization: helpful perspective on recommendation, and also in other settings (PCA)

Graph models: generalisation of recommendation, apply **graph convolution** to look deeper into the graph.

82

mlcourse@peterbloem.nl